Because a rational number is determined by the integer numerator and integer denominator, we defined our data structure for a rational number as

```plaintext
type rational
  integer :: num
  integer :: den
end type rational
```

Our goal is to add functionality to our module so that we can add or subtract two rational numbers, multiply two rational numbers, reduce a rational number to its simplest form, print out a rational number, convert it to a floating point number, etc.

We first added a subprogram to add two rational numbers.
function add_two_rationals (a, b )  result (c)
  type(rational), intent(in) ::  a, b
  type(rational) ::  c
  
c % num = a % num * b % den + b % num * a% den
  c % den = a % den * b % den
end function add_two_rationals

• Next we saw that we can use operator overloading to simplify addition of two rational numbers.

• To do this we added an interface construct

  interface operator (+)
    module procedure add_two_rationals
  end interface

• Now in the main program we can simply type

  \[ c = a + b \]

  when \( a,b,c \) are declared as the data structure rational. This statement
is the same as

\[ c = \text{add\_two\_rationals}(a, b) \]

- We can also write a routine to subtract two rational numbers and overload the operator \(-\). You will be asked to do this for classwork.
- Before we try out the code, let's briefly look at how we can write a routine to reduce a rational number to its simplest fractional form.
- For example, if we add

\[ \frac{1}{10} + \frac{7}{10} \]

then our routine will return \( \frac{8}{10} \) instead of \( \frac{4}{5} \).
- Basically, we want to find the greatest common divisor (gcd) or equivalently the greatest common factor. In school we were taught to write the factors of each number and then pick the largest common factor. Can we translate this into an algorithm we can program? Is there some other way to find the gcd without listing all the divisors of each number?
\[
\frac{8}{10} = \frac{2 \cdot 4}{2 \cdot 5} = \frac{4}{5}
\]

- It is typically much easier to form a division than to find the factors of a number. Consequently, we choose a method called Euclid’s algorithm which uses the division method in combination with the observation that the gcd of two numbers also divides their difference. Euclid’s proof was geometric in nature because algebra had not been created yet.

- The division theorem just says that if we have two integers \( p, d \) then

\[
p = qd + r
\]

where \( d \) is the divisor, \( q \) the quotient and \( r \) the remainder; moreover \( q, r \) are unique; i.e.,

\[
\frac{p}{d} = q + \frac{r}{d}
\]
• Consider the example $\frac{35}{25}$. From experience, we know that the gcd is 5. We note that

$$\frac{35}{25} = 1 \text{ with a remainder of } 10$$

and

$$\frac{25}{10} = 2 \text{ with remainder of } 5$$

and

$$\frac{10}{5} = 2 \text{ with remainder of } 0$$

so the gcd is 5 because the last nonzero remainder is our gcd. Basically we have written

$$35 = 25 + 10 = (2 \cdot 10 + 5) + 10 = 2(2 \cdot 5) + 5 + (2 \cdot 5)$$

• If we have a rational like $\frac{7}{3}$ which is already in its simplest form, does the above algorithm work?
\[ \frac{7}{3} = 2 \text{ with a remainder of } 1 \]

and

\[ \frac{3}{1} = 3 \text{ with remainder of } 0 \]

so the gcd is 1.

If we have a number like \( \frac{9}{3} \) then we are done on the first step since immediately we have a remainder of zero and we know 3 is our gcd.

Euclid’s algorithm for finding the gcd

Given integers \( p, d, d \neq 0 \). Set \( r = 1 \) arbitrarily

while \( r \neq 0 \)

calculate remainder \( r \) from \( \frac{p}{d} \)

if \( r = 0 \) then \( \text{gcd} = d \) else \( p = d, d = r \)
What do we need to implement this algorithm?

First we need a function which calculates the remainder when we do integer arithmetic. Fortran has the intrinsic function \texttt{modulo} where \texttt{modulo} (p,q) returns an integer which is the remainder of the division \( \frac{p}{q} \); for example \texttt{modulo} (35,25)=10.

We need to define a function which calculates the remainder of \( \frac{p}{q} \), i.e., \texttt{modulo}(p,q), repeatedly. The first call \( p,q \) are set by the numerator and denominator of the fraction to be reduced. On the second call \( q \) is the numerator and the denominator is the remainder from the previous step. We continue until the remainder is zero. The input is two integers and the output an integer.

Fortran has something called a \textbf{recursive function} to do this. This allows the function to call itself. When using a recursive function we must use the specifier \texttt{result}. Then we would have a subroutine, say \texttt{reduce} which would call this recursive function. In our function \texttt{add_two_rationals} we have included a call to \texttt{reduce}. Care must be used with recursive routines since it is easy to get into an infinite loop.
recursive function gcd (p, q ) result ( div )! must use result
integer :: div ! greatest common divisor on return
integer, intent(in) :: p,q ! numerator and denominator of rational number a on first call
if ( q == 0 ) then
    div = p
else
    div = gcd ( q, modulo (p, q ) )! recursive part of function
end if
end function gcd

• Consider our example of $\frac{35}{25}$.
  – On first call to gcd p = 35, q = 25
  – Since q is not zero, we call gcd again with p=25, q=10 (10 is the remainder.
Since \( q \) is not zero, we call \( \gcd \) again with \( p=10, \ q=5 \) (5 is the remainder.

Since \( q \) is not zero, we call \( \gcd \) again with \( p=5, \ q=0 \) (0 is the remainder.

The \( \gcd \) is set to 5.

• Consider our routine \texttt{reduce} which calls the function \( \gcd \) and then divides the numerator and denominator by the \( \gcd \). Should it be a function or a subroutine? What is its input and what is its output?

• We want to input a member of the data structure \texttt{rational} and output another member of the data structure. However, we don’t need to keep the input, we can simply overwrite it. So the input and output is the same member of our data structure, we have simply overwritten its numerator and denominator.
subroutine reduce (a)

type(rational) :: a

integer :: div

div = gcd ( a % num, a % den )! recursive function

a% num = a% num / div

a% den = a% den / div

end subroutine reduce

In our routine to add two rational numbers we will add a call to reduce so that after we add the numbers we will then reduce to its simplest form.

We will declare reduce, gcd private because they are only used by other routines in our module.
We can easily add a routine to multiply two rational numbers and overload the operator \(*\).

```fortran
function multiply_two_rational(a, b) result(c)
  type(rational), intent(in) :: a, b
  type(rational) :: c
  c % num = a % num * b % num
  c % den = a % den * b % den
  call reduce(c)
end function multiply_two_rational
```

```fortran
interface operator (*)
  module procedure multiply_two_rational
end interface
```
Completing our Rational Number Class

- Our ultimate goal for Part III is to write a complete vector class and think about a sparse vector class. Before we move ahead to learn about vectors, we want to see what it means to write a (fairly) complete class using our simple rational number example.

- So far we have the derived data type `rational` to define our data structure, functions to add and multiply two rationals where we have overloaded the corresponding operators `+` and `*`. In addition, we have included two internal routines for use in reducing a rational to its simplest form.

- In a straightforward manner we can write functions to subtract and divide two rational numbers and overload the corresponding operators.

- To complete our module we need to add a constructor and destructor and then add subprograms for all the operations we can think of that might be needed.
- For the destructor (say `delete_rational`), we can simply set the rational number to zero using either the manual approach `a % num = 0; a % den = 1` or the intrinsic constructor `a = rational(0,1)`.

- For the constructor (say `make_rational`), we could have the integers `numerator, denominator` as optional input. If they are present (and the denominator is not zero) then we set the rational number’s numerator and denominator to these values; e.g., `a % num = numerator`, etc. If not present set to a default value of zero, i.e., `rational(0,1)`.

- We could add a routine to print out a rational number in a format such as `7/2`.

- We could add a routine to convert a member of our `rational` data structure to floating point using either `float` or `dfloat`.

- Another routine that might be useful is to invert a rational. In fact, a simple way to write a routine to divide two elements of our `rational` data structure would be to have a call to invert one element and then call the routine to multiply two elements of our data structure.

- We could add a routine to multiply a rational by an integer and output the
result in a new member of the `rational` class. Note that if we multiply a rational number by an arbitrary floating point number the result is not, in general, a rational number. This is an easy routine to write but can we also overload the operator `*` to do this because we have already overloaded it to call `multiply_two_rationals`? Assume our routine has the function statement

```fortran
function multiply_scalar_rational (a,b) result(c)
where we have declared type(rational), intent(in) ::  a
integer, intent(in) ::  b
type(rational) ::  c
and we add the interface statement

interface operator (*)
   module procedure multiply_two_rationals, multiply_scalar_rational
end interface
```

If `a`, `c`, `d` are declared as rationals and `b` an integer then what will each statement do?
\[ d = a \times c \]
\[ d = a \times b \]
\[ d = b \times a \]

- Sometimes we need a temporary copy of a member of our class. For example, if we wanted to write a routine to check if two rational numbers are equal, then we would want to reduce them first and then see if their numerators are equal and their denominators are equal. However, our reduce routine overwrites the rational number with the simplified one so if we don’t want to destroy the rational numbers on input (to the routine to check if numbers are equal), then we must make a temporary copy. So a routine to copy a rational into another rational is useful as well as checking to see if two members of our rational data structure are identical (more on this in a minute).

- In writing each routine you need to first think carefully about whether your routine should be a function or subroutine. Remember this is governed by whether or not you have a single output to be returned to the calling unit. If it is a single output, we normally write a function; otherwise we write a subroutine. If there are multiple outputs then you must use a subroutine. Remember that a member of a data structure is considered a single output.
even though it may contain several different variables.

- Now let's look at the routines to define overloaded other operations and assignments.

- We already have a routine to overload “+” and “*”. We could overload “-” by writing a routine analogous to `add_two_rationals` or one which calls multiplication by a scalar (with the scalar set to -1) and then calls the routine to add the two rational numbers. Similarly for the division of two rationals we can write a routine to first invert and then multiply. When we write both of these routines we do not want to destroy the input members of our rational class so we may need to make a copy first. For example, we could have the following routine for making a copy

```fortran
function copy_rational (a) result(temp)
  type(rational) :: a, temp
  temp = a
end function copy_rational
```

then the routine for dividing two rational numbers might have a calling statement

```fortran
function divide_two_rationals ( a, b ) result (c)
```

with a section of code like

```fortran
b_temp = copy_rational (b)
call invert_rational(b_temp)
c = a * b_temp
```

This way we do not overwrite `b`. Of course in a routine for dividing, we should always check to make sure we are not dividing by a zero rational number.

- Now let's consider overloading the assignment `“=”`. In this case we want to write `a = 2` to mean we are setting the rational number to `rational(2,1)` using the intrinsic constructor. We are going to use a subroutine here so that you can see that you can overload an operator with a subroutine as well as a function. Note that here the member of your data structure has to be the first argument. It won't work if you interchange the order of the arguments.

  - First we need to declare our interface statement where now we use `assignment` instead of `operator`

    ```fortran
    interface assignment (=)
      module procedure set_equal_integer
    end interface
    ```
end interface

- Our subroutine could simply look like

```fortran
subroutine set_equal_integer ( new, integer_value ) ! order of arguments matters
type(rational), intent(out) :: new
integer, intent(in) :: integer_value

new % num = integer_value
new % den = 1

end subroutine set_equal_integer
```

- Then in the main program we simply say, for example, `a = 2` to invoke this subroutine rather than `call set_equal_integer (a, 2)`.

- If you reverse the order of the arguments in the subroutine statement to

```fortran
subroutine set_equal_integer ( integer_value, new )
```

you will get an error saying it is incorrect for a defined assignment.
Lastly we will overload the operators "==", ">" and "<". Because we plan to use these in a conditional like testing if a equals b or a greater than b, we want the result of this function to be a logical. So, for example, to overload ">" we have the interface

```
interface operator (> )
    module procedure is_greater_than
end interface
```

and our routine could simply convert each rational to a real number and then compare. For example,

```
function is_greater_than (a,b ) result (t_f)
type(rational), intent(in) ::  a,b
logical ::  t_f
real (prec) ::  a_real, b_real

a_real = convert_rational_real (a )
b_real = convert_rational_real (b )
t_f=.false.
if ( a_real > b_real ) t_f=.true.
```
end function is_greater_than

- Recall that when we set a logical the syntax requires us to use a period before and after, i.e., \texttt{t.f=.false.}.

- The function to overload \texttt{<} is similar.

- To overload the operator \texttt{==} we simply check to see if two rational numbers are equal; for example, is \( \frac{2}{3} = \frac{4}{6} \)? To do this we must first reduce each to its simplest form (remember that we should not destroy the input) and then check to see if their numerators are equal and their denominators are equal. We already have a routine to reduce a rational to its simplest form so this should be straightforward.

- For your next homework you are asked to complete our module for rational numbers. Your strategy should be to add one capability and test it completely before moving on to the next. Do not try to implement all the routines at once.
Classwork

- Make files - these can save you time when we have several .o files to add to the command line.

- The file class_rational.f90 contains the module class_rational with the data type rational defined, the functions add_two_rationals, multiply_two_rationals and with the statements to overload the +, * operators. In addition, it contains the private routines reduce and gcd. The main program test_rational.f90 which defines two rationals (a, b) and adds them with the statement c = a + b and multiplies them with c = a * b. Try this to make sure it works and reduces the number to its simplest form.

- Try to call the subroutine reduce(a) in the main program to see the error you get.

- Modify the module to add a routine to multiply a rational with a scalar and overload the operator * . Add statements in the driver program to test this.
Add a routine to subtract two rational numbers and reduce it to simplest form by using the existing routines of scalar multiplication and addition. Add statements to overload the operator \(-\). Add statements in the driver program to test this.
Homework

• HwkIII_2, due Friday, February 28